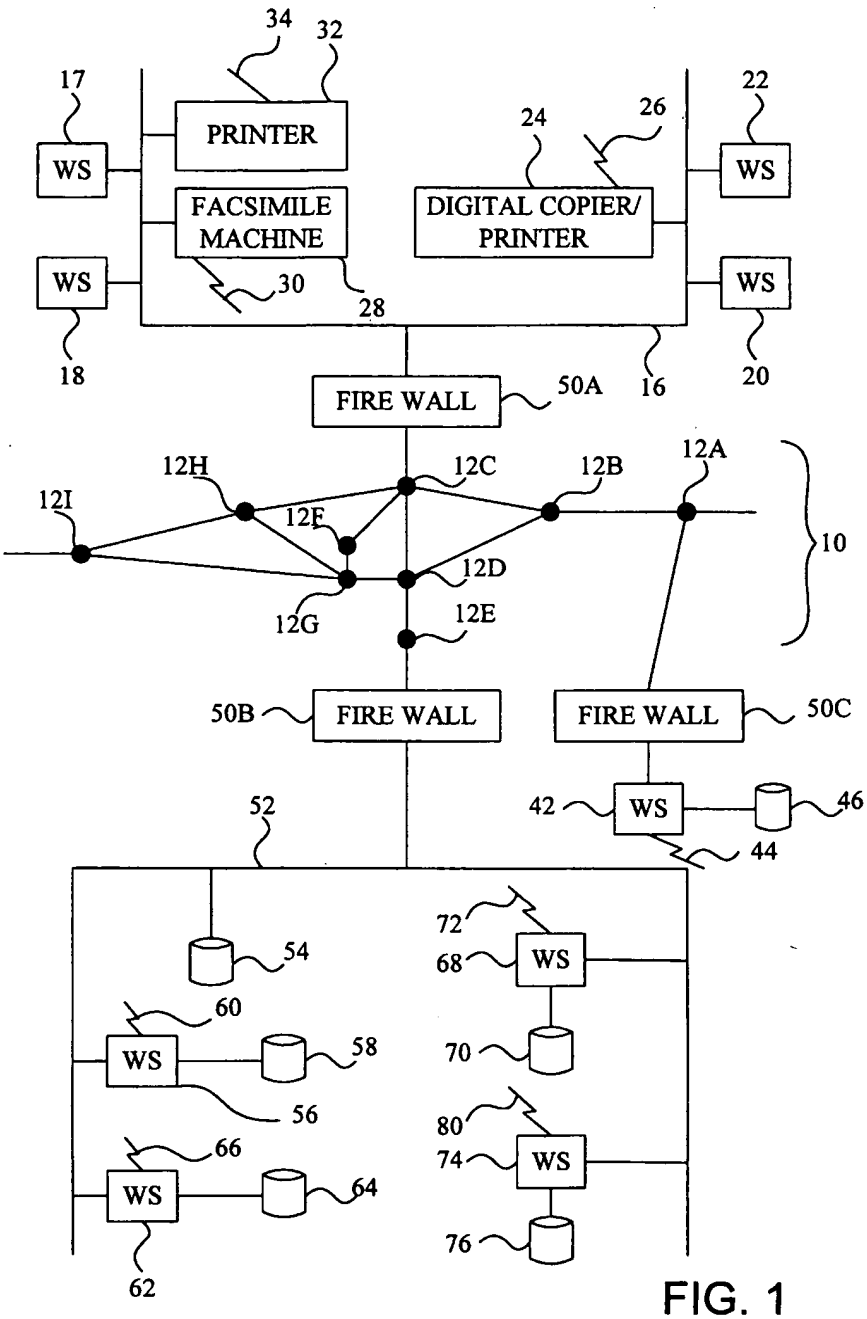

34
32
17
PRINTER
22
24
26
WS
FACSIMILE MACHINE
DIGITAL COPIER/ PRINTER
WS
WS
30
28
WS
18
16
20
FIRE WALL
50A
12H
12C
12B
12A
12I
12F
10
12D
12G
12E
50B
FIRE WALL
FIRE WALL
50C
52
42
WS
46
44
54
72
68
WS
60
WS
58
70
56
80
66
74
WS
WS
64
76
62

FIG. 1

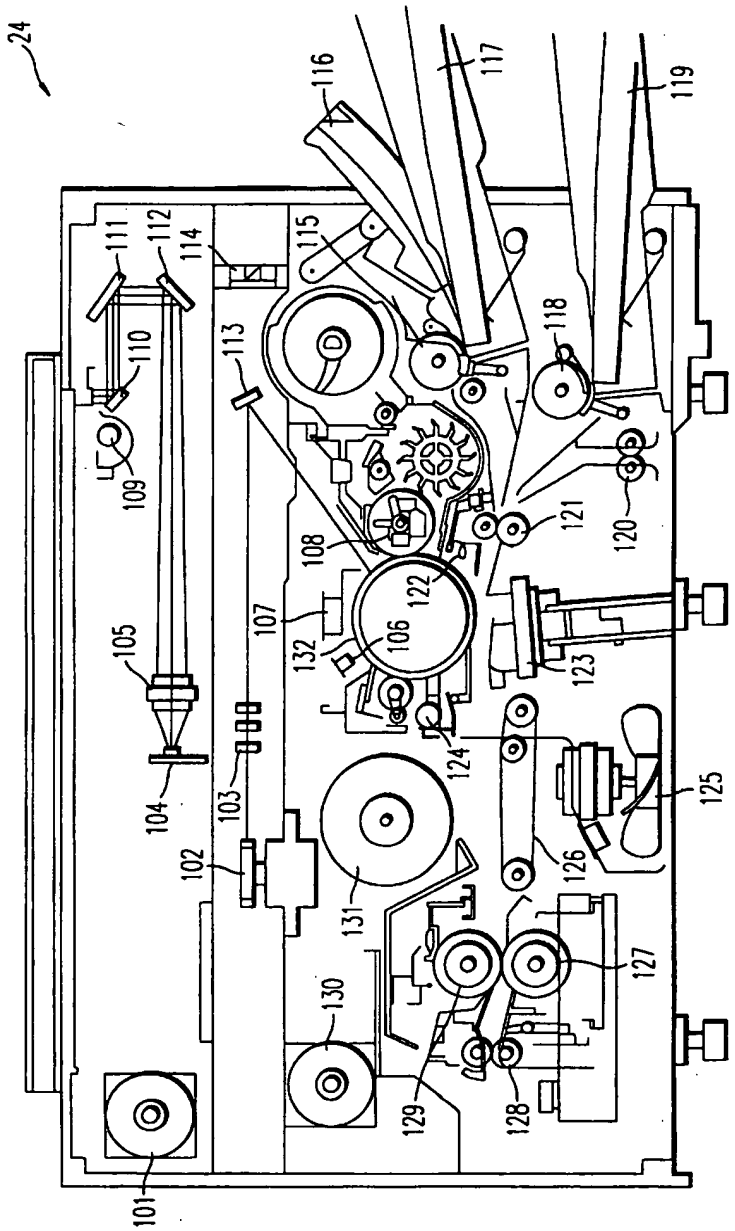

24
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132

FIG. 2

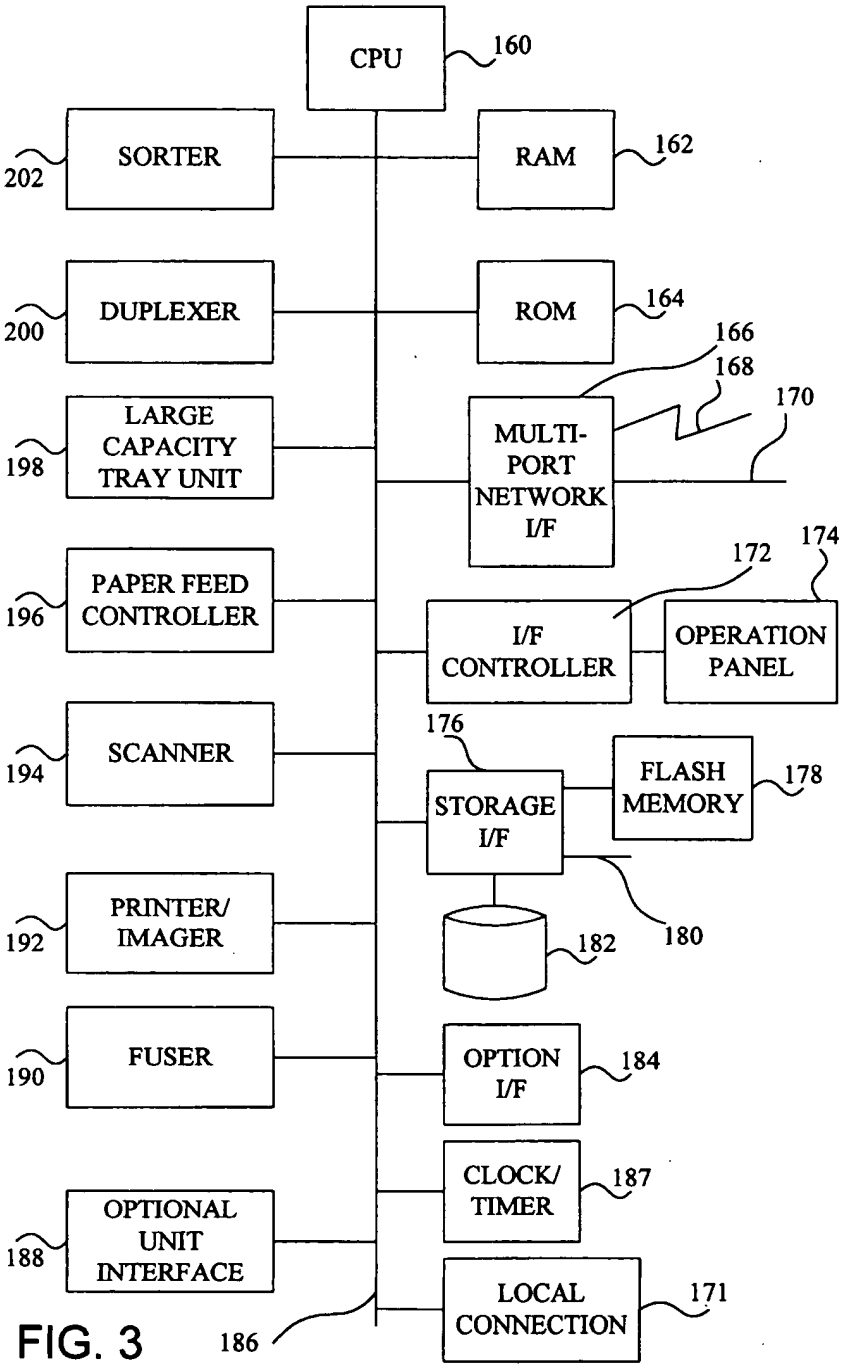

CPU
160
SORTER
202
RAM
162
DUPLEXER
200
ROM
164
166
168
170
LARGE CAPACITY TRAY UNIT
198
MULTI-PORT NETWORK I/F
PAPER FEED CONTROLLER
196
172
174
I/F CONTROLLER
OPERATION PANEL
SCANNER
194
176
FLASH MEMORY
178
STORAGE I/F
PRINTER/ IMAGER
192
182
180
FUSER
190
OPTION I/F
184
CLOCK/ TIMER
187
OPTIONAL UNIT INTERFACE
188
LOCAL CONNECTION
171
186

FIG. 3

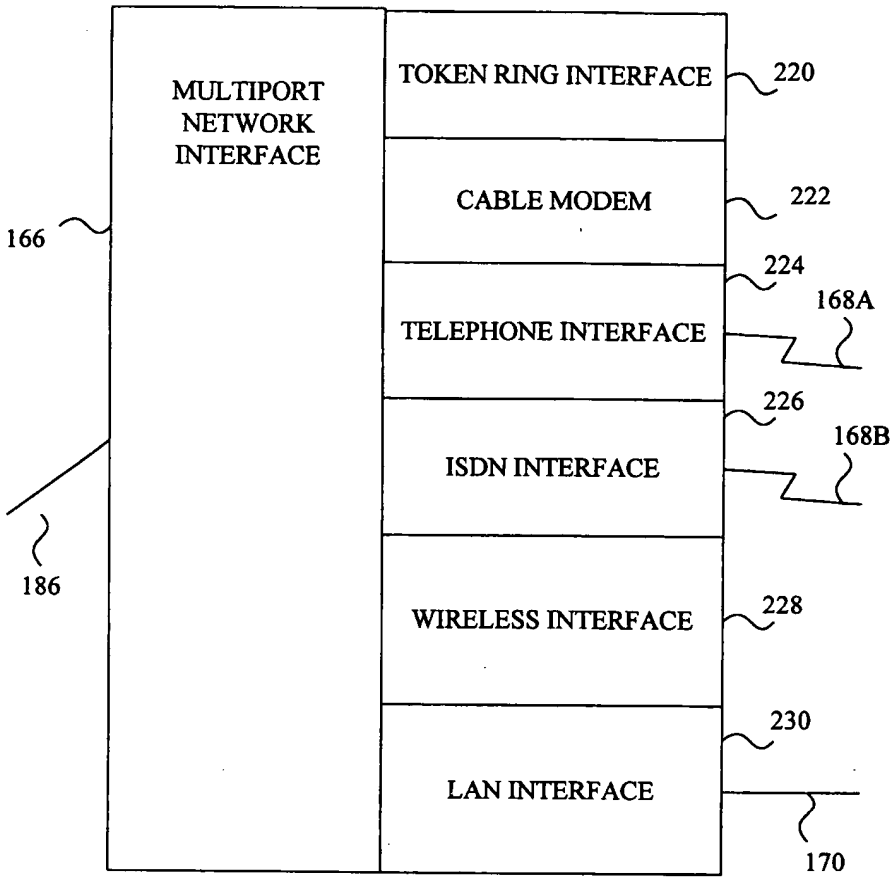

MULTIPORT NETWORK INTERFACE
166
186
TOKEN RING INTERFACE
220
CABLE MODEM
222
224
TELEPHONE INTERFACE
168A
226
ISDN INTERFACE
168B
WIRELESS INTERFACE
228
230
LAN INTERFACE
170

FIG. 4

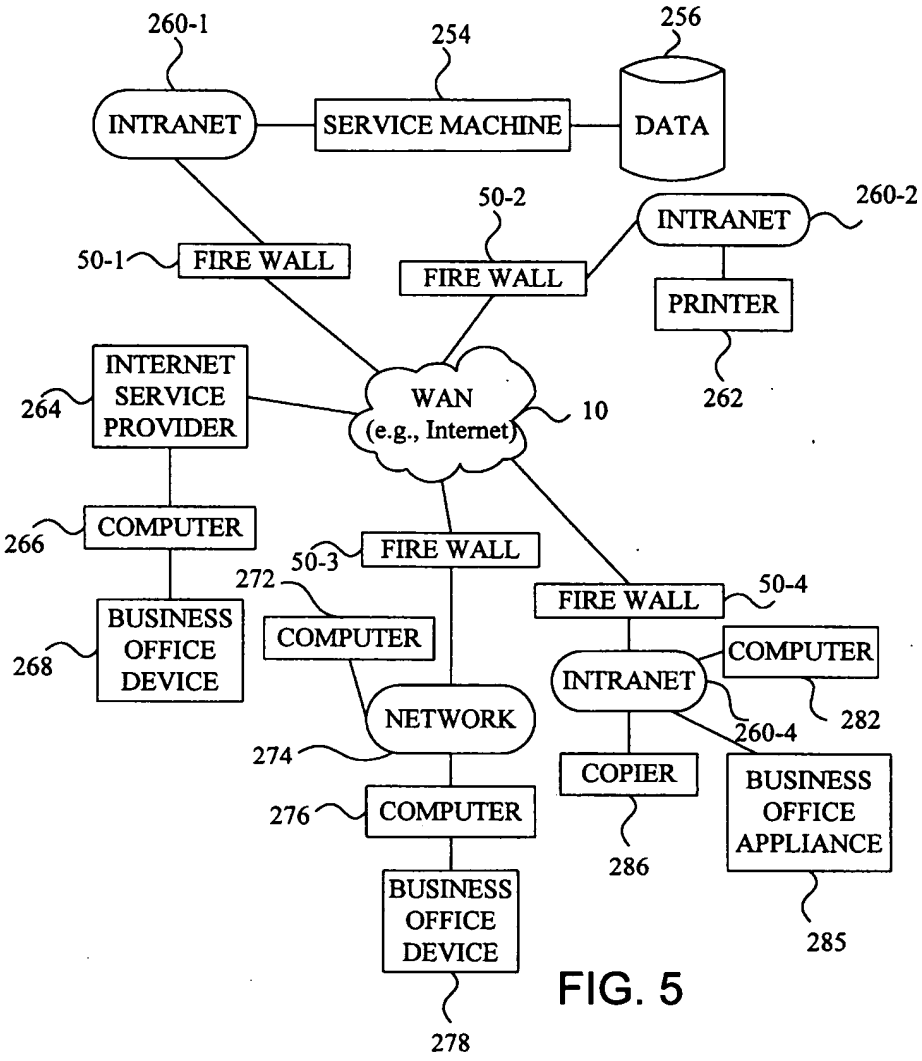


FIG. 5

09453535-050400

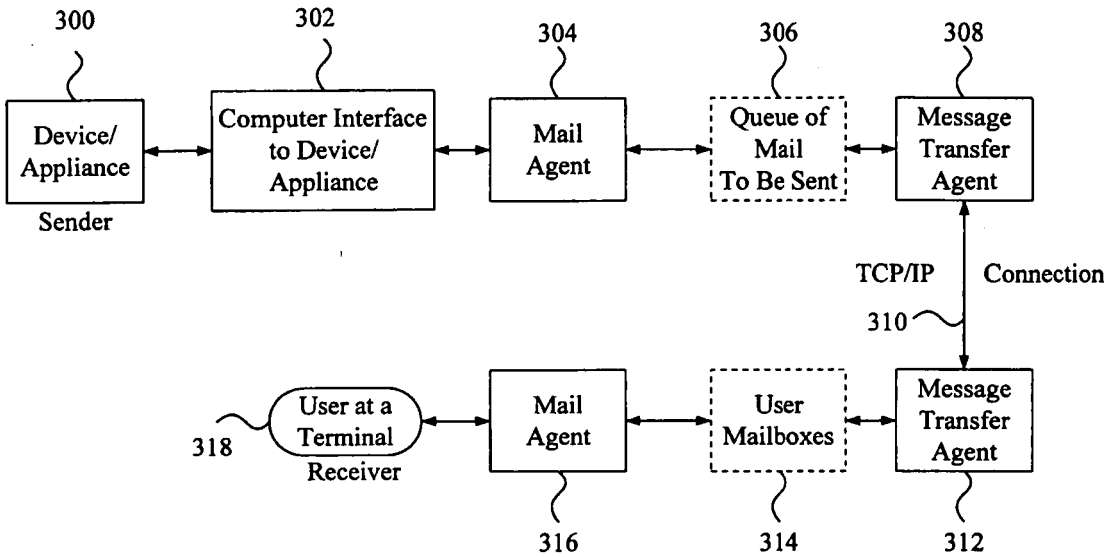

300
302
304
306
308
Device/ Appliance
Sender
Computer Interface to Device/ Appliance
Mail Agent
Queue of Mail To Be Sent
Message Transfer Agent
TCP/IP
Connection
310
318
User at a Terminal
Receiver
Mail Agent
User Mailboxes
Message Transfer Agent
316
314
312

FIG. 6A

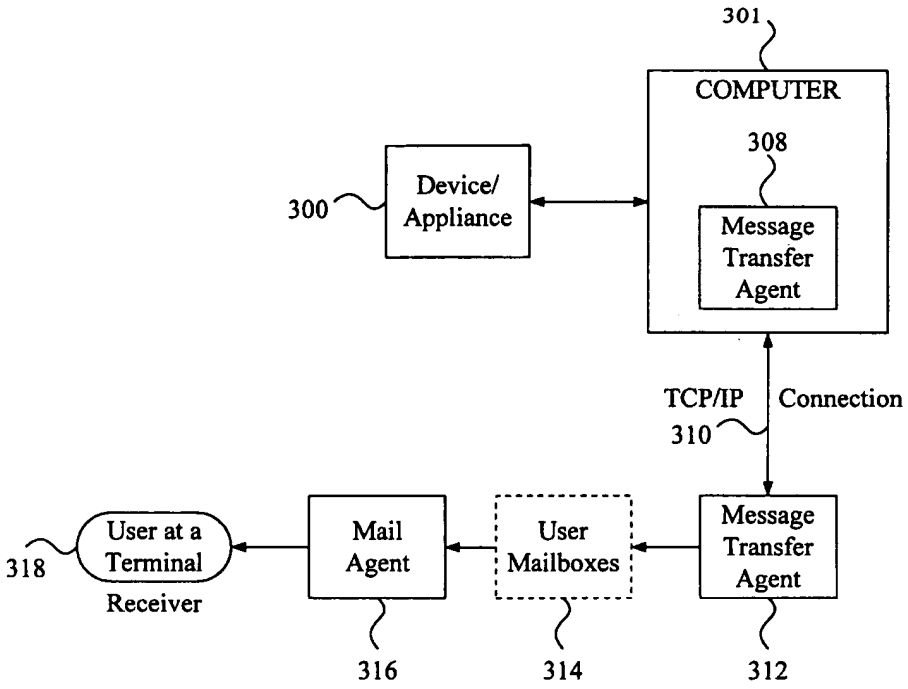

301
COMPUTER
308
Message Transfer Agent
300
Device/ Appliance
TCP/IP Connection
310
Message Transfer Agent
User Mailboxes
Mail Agent
User at a Terminal
318
Receiver
316
314
312

FIG. 6B

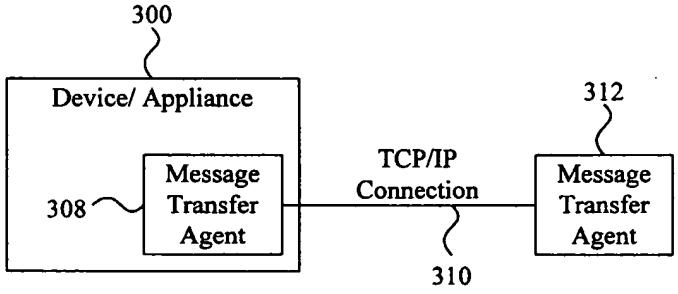

300
Device/ Appliance
308
Message Transfer Agent
TCP/IP Connection
310
312
Message Transfer Agent

FIG. 6C

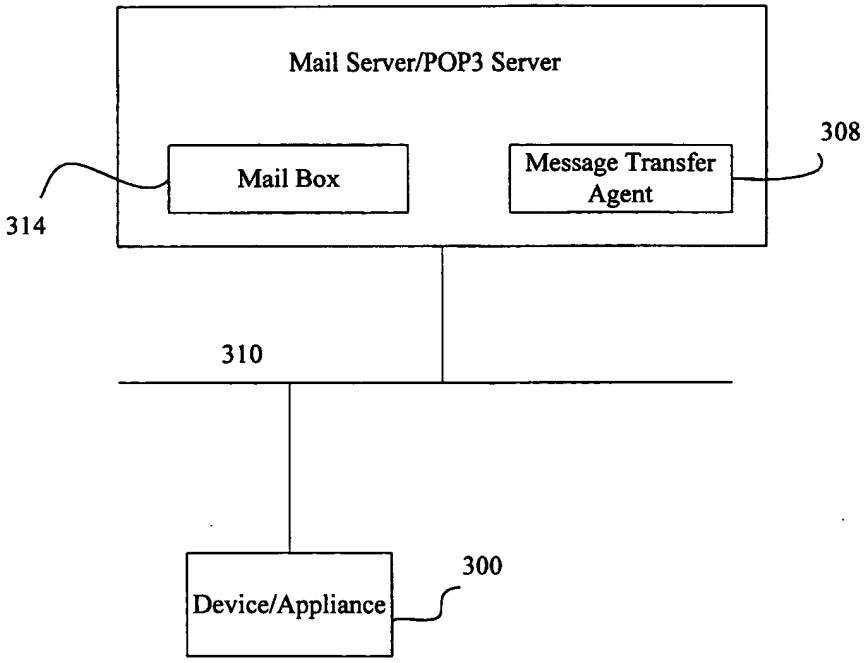


Figure 6D

09/754,938 - 05/03/2004

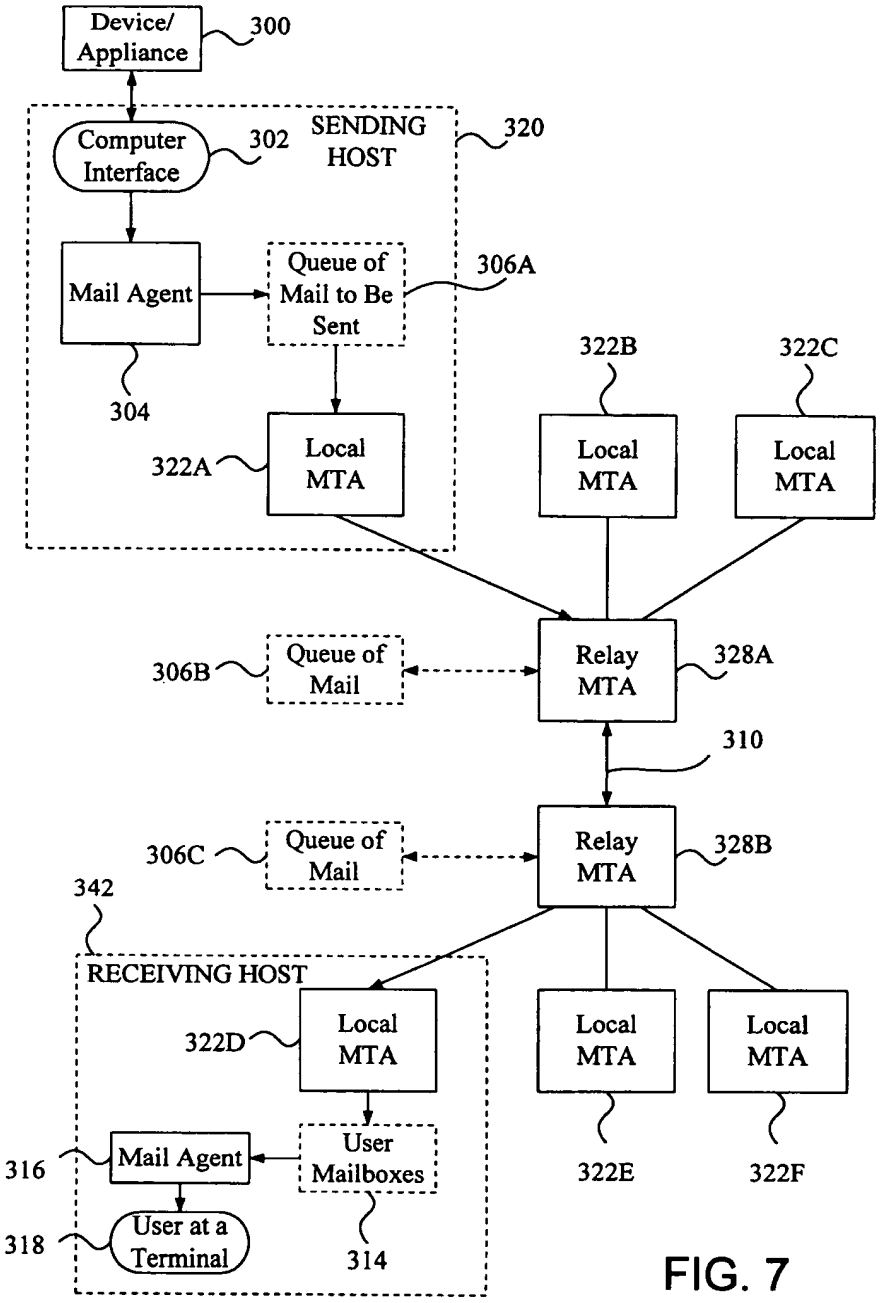

Device/ Appliance
300
Computer Interface
302
SENDING HOST
320
Mail Agent
Queue of Mail to Be Sent
306A
304
322A
Local MTA
322B
Local MTA
322C
Local MTA
306B
Queue of Mail
Relay MTA
328A
310
306C
Queue of Mail
Relay MTA
328B
342
RECEIVING HOST
322D
Local MTA
Local MTA
Local MTA
316
Mail Agent
User Mailboxes
318
User at a Terminal
314
322E
322F

FIG. 7

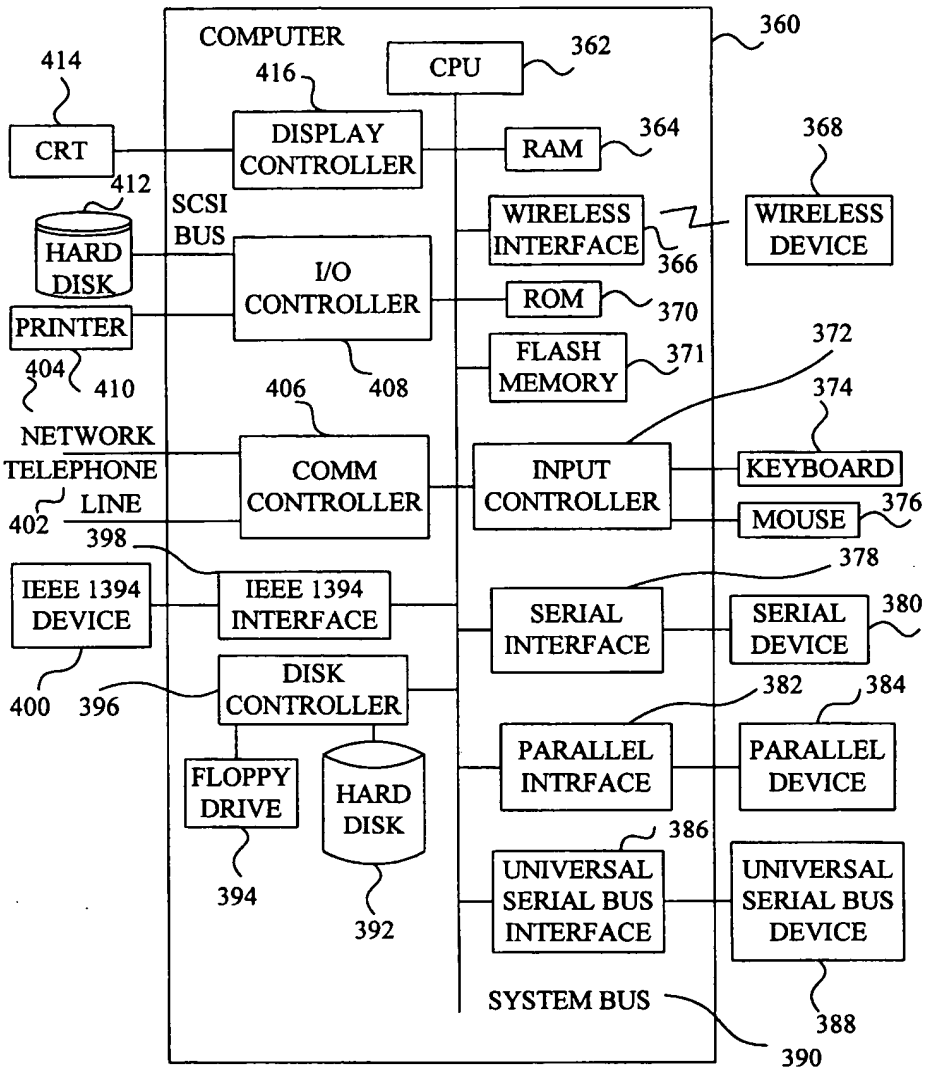

COMPUTER
360
CPU
362
414
416
CRT
DISPLAY CONTROLLER
RAM
364
368
412
SCSI BUS
WIRELESS INTERFACE
366
WIRELESS DEVICE
HARD DISK
I/O CONTROLLER
ROM
370
PRINTER
FLASH MEMORY
371
372
404
410
408
406
374
NETWORK TELEPHONE LINE
COMM CONTROLLER
INPUT CONTROLLER
KEYBOARD
402
MOUSE
376
398
378
IEEE 1394 DEVICE
IEEE 1394 INTERFACE
SERIAL INTERFACE
SERIAL DEVICE
380
400
396
DISK CONTROLLER
382
384
FLOPPY DRIVE
HARD DISK
PARALLEL INTRFACE
PARALLEL DEVICE
386
394
392
UNIVERSAL SERIAL BUS INTERFACE
UNIVERSAL SERIAL BUS DEVICE
SYSTEM BUS
388
390

FIG. 8

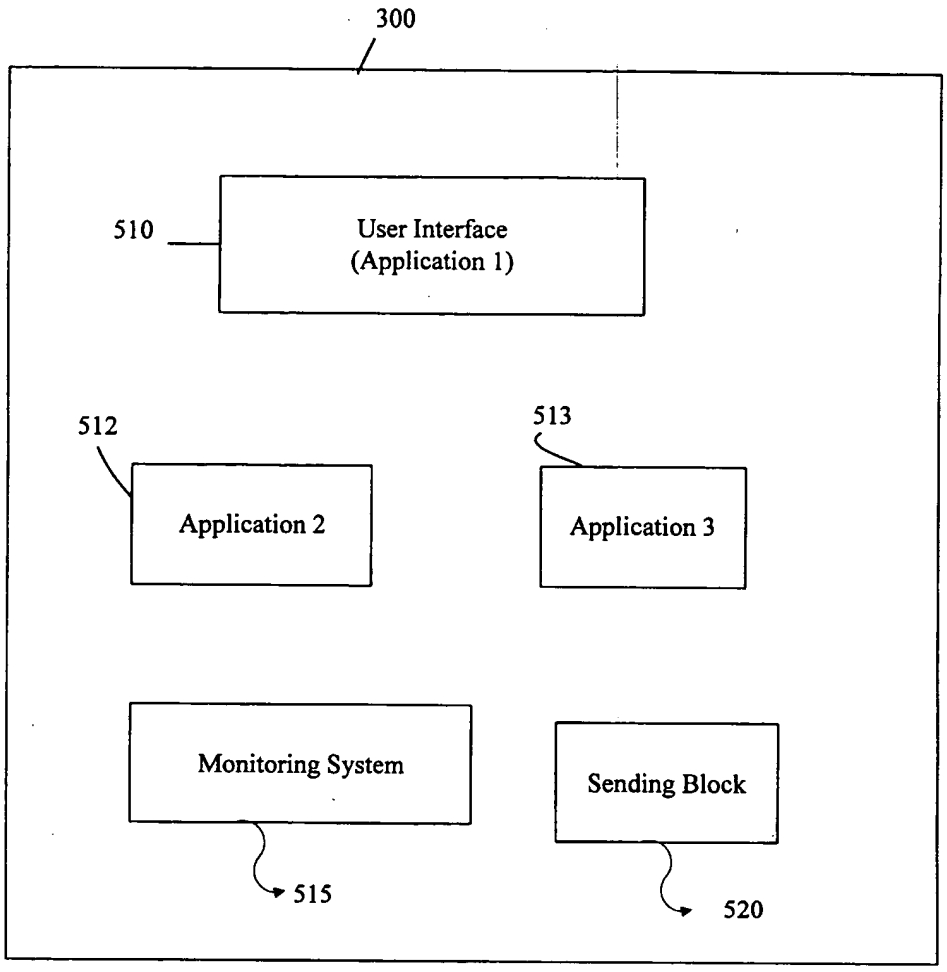

300
510
User Interface
(Application 1)
512
Application 2
513
Application 3
Monitoring System
515
Sending Block
520

Fig. 9

Figure 10

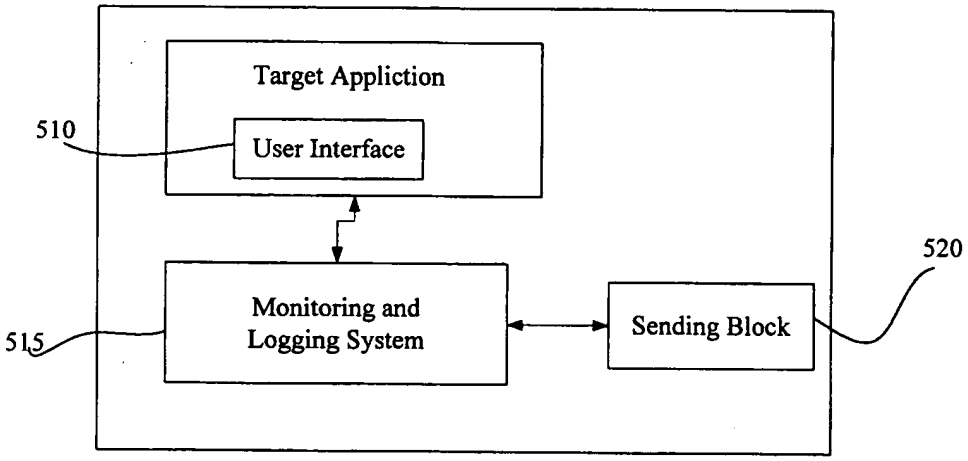
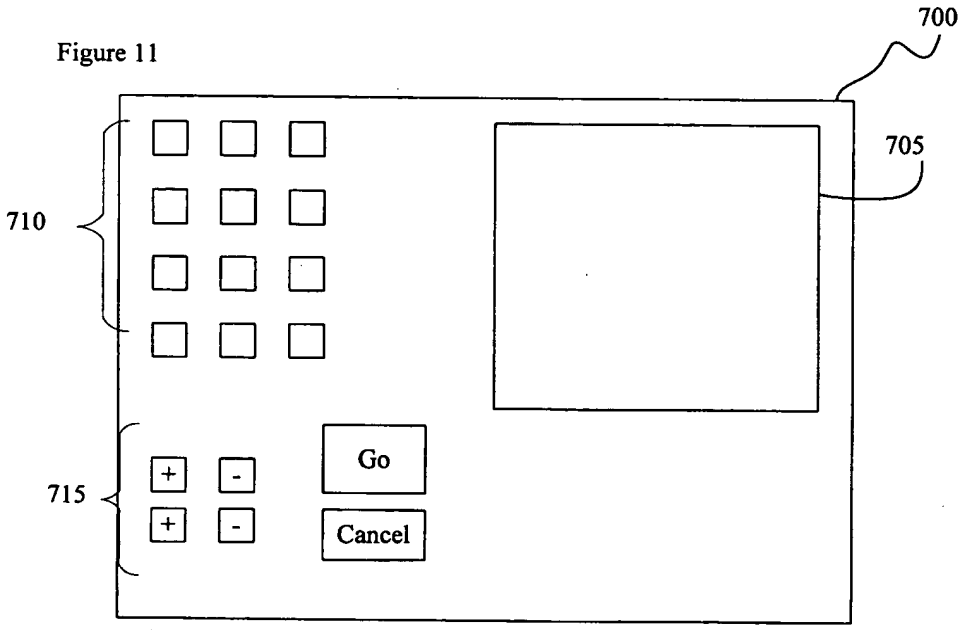


Figure 11



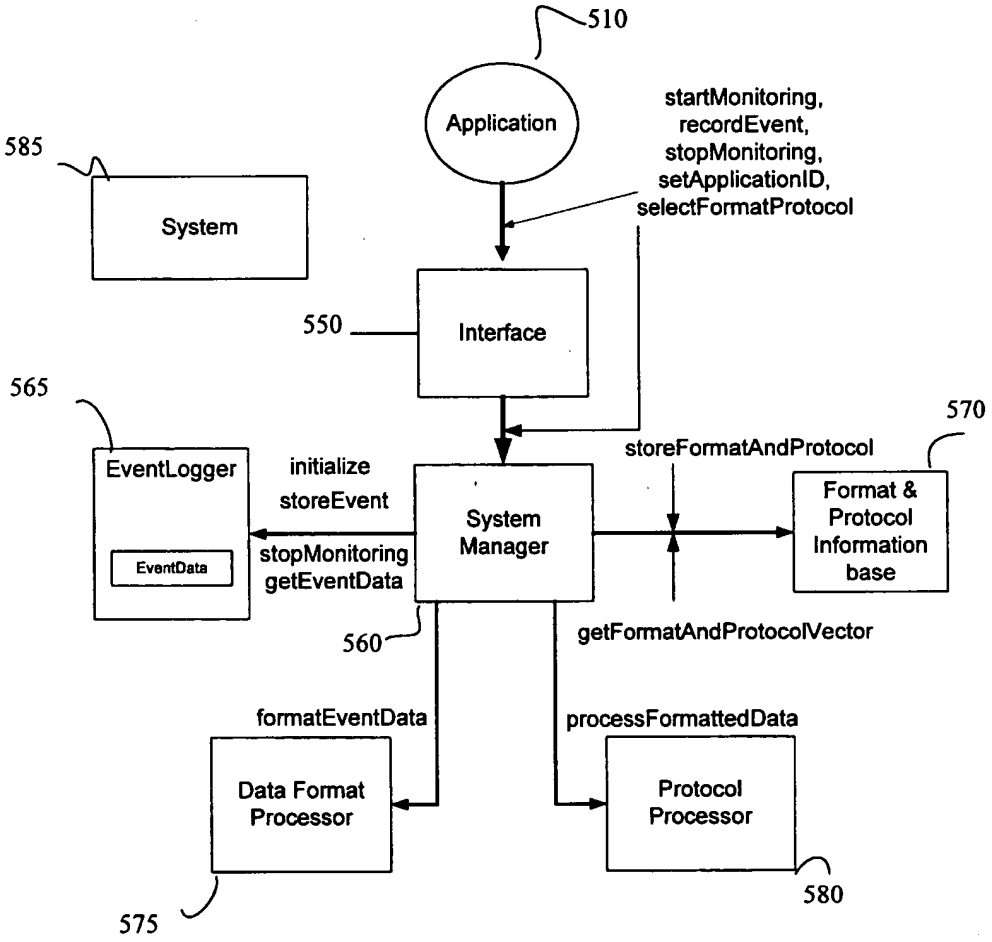

510
Application
startMonitoring,
recordEvent,
stopMonitoring,
setApplicationID,
selectFormatProtocol
585
System
550
Interface
565
570
EventLogger
EventData
initialize
storeEvent
stopMonitoring
getEventData
System
Manager
storeFormatAndProtocol
Format &
Protocol
Information
base
560
getFormatAndProtocolVector
formatEventData
processFormattedData
Data Format
Processor
Protocol
Processor
575
580

Figure 12A

Return Value	Function Name	Description
bool	getNextSession	Returns false when there is no more session; true otherwise
string	getFileName	Returns file name for the EventData
map<string, string>	getSessionInformation	Returns the map. Keys are UserID, ApplicationID, CumulativeSessionNumber, StartTime, and Duration.
map<string, vector<string>>	getSessionEventData	Returns the map. Keys are EventName and EventTiming. The values of EventTiming vector are in the unit of 10th of a second converted from unsigned integer to string.

Figure 12B

Return Value	Function Name	Description
bool	getNextLine	Returns one line of string data as an out parameter string. The function returns true if there is a line; false if no more line exists with empty string.
string	getFileNameWithSuffix	Returns file name for the data with suffix if applicable

Figure 12C

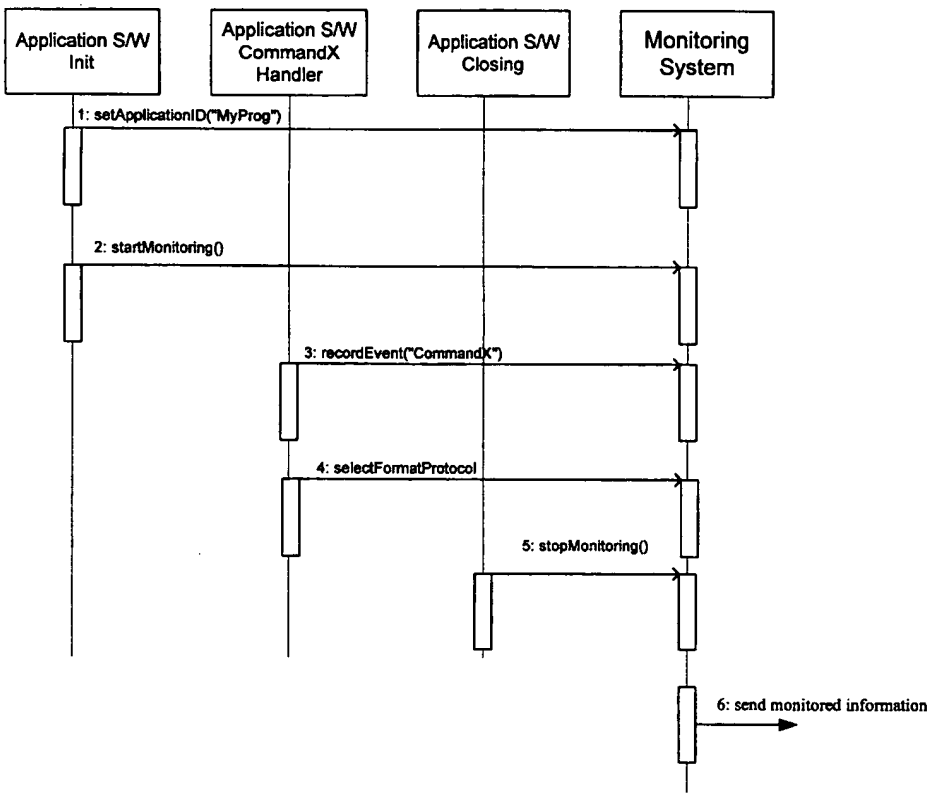

Application S/W Init
Application S/W CommandX Handler
Application S/W Closing
Monitoring System
1: setApplicationID("MyProg")
2: startMonitoring()
3: recordEvent("CommandX")
4: selectFormatProtocol
5: stopMonitoring()
6: send monitored information

Figure 13

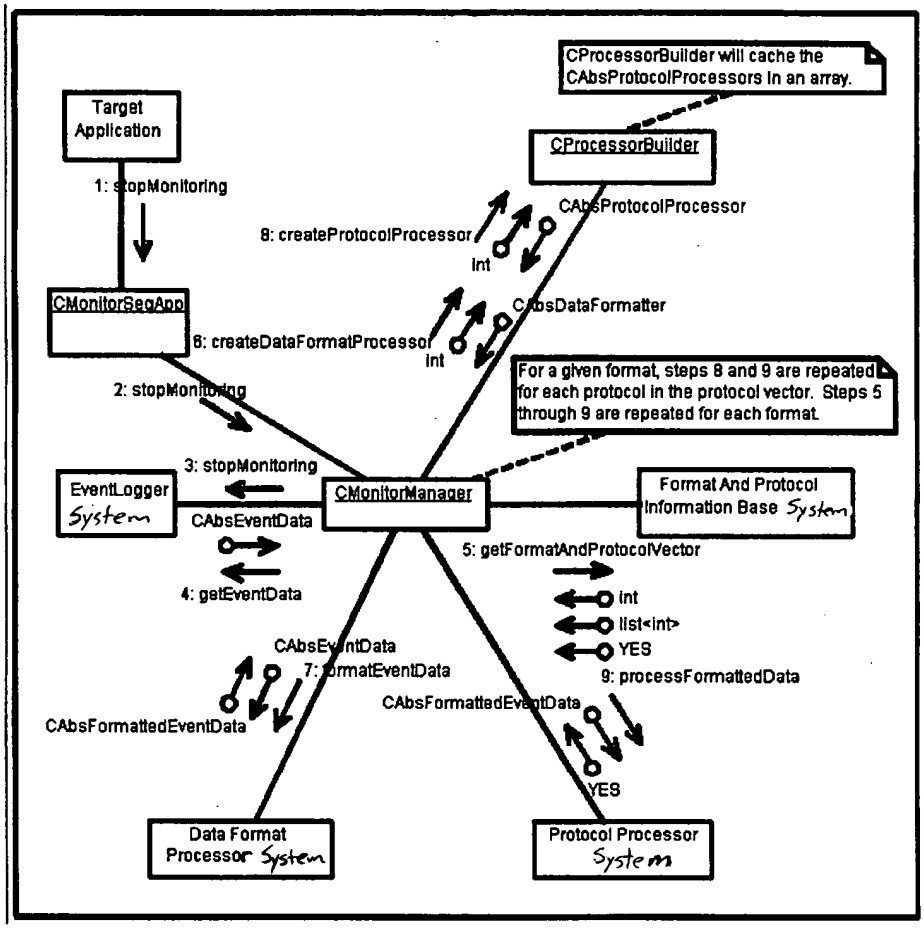

CProcessorBuilder will cache the CAbsProtocolProcessors in an array.
Target Application
CProcessorBuilder
1: stopMonitoring
CAbsProtocolProcessor
8: createProtocolProcessor
int
CMonitorSegApp
CAbsDataFormatter
6: createDataFormatProcessor
int
For a given format, steps 8 and 9 are repeated for each protocol in the protocol vector. Steps 5 through 9 are repeated for each format.
2: stopMonitoring
3: stopMonitoring
EventLogger System
CMonitorManager
Format And Protocol Information Base System
CAbsEventData
5: getFormatAndProtocolVector
4: getEventData
int
list<int>
YES
CAbsEventData
7: formatEventData
9: processFormattedData
CAbsFormattedEventData
CAbsFormattedEventData
YES
Data Format Processor System
Protocol Processor System

Figure 14

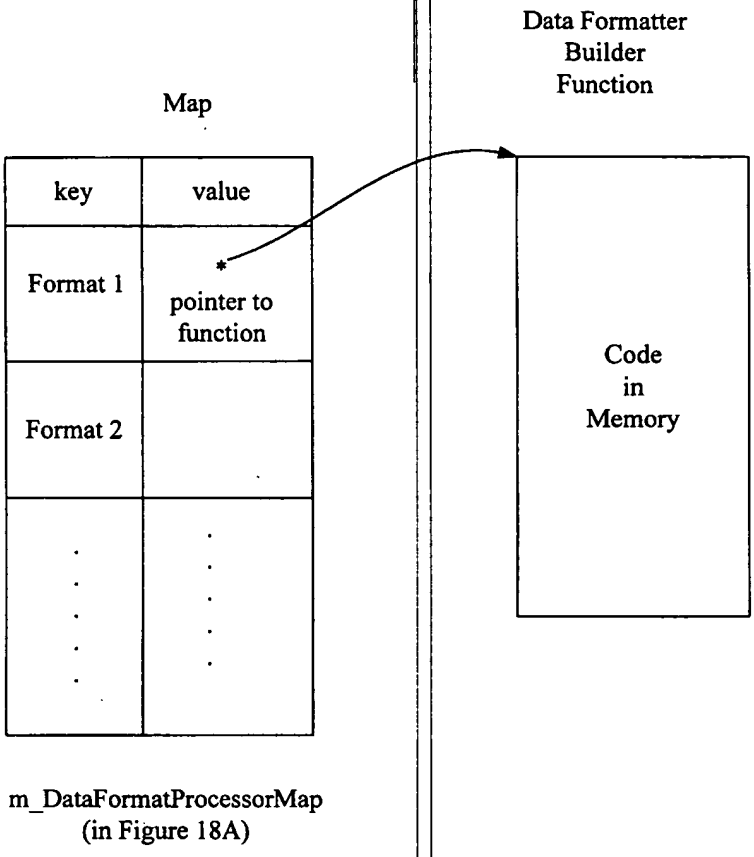

Data Formatter
Builder
Function
Map
key
value
Format 1
pointer to
function
Format 2
Code
in
Memory
m_DataFormatProcessorMap
(in Figure 18A)

Figure 15

```
void CMonitorManager::stopMonitoring()
{
	TRACE("CMonitorManager::stopMonitoring \n");

// 1.  calls the function stopMonitoring() of
//     CUsageLogger.
	m_UsageLogger.stopMonitoring();

// 2.  calls the function getEventData() of
//     CUsageLogger.  This function returns the usage
//     information, CAbsEventData, to CMonitorManager.
	CAbsEventData * loc_pAbsEventData = m_UsageLogger.getEventData();

// 3.  calls the function getFormatAndProtocolVector()
//     of CFormatProtocol_InformationBase.  This function
//     returns the following to CMonitorManager: an int for
//     the data format, a list<int> for the communication
//     protocols, and a bool to indicate if the return
//     values (format and protocol) are valid.

	int loc_nFormat;
	list<int> loc_ProtocolVector;

	CProcessorBuilder loc_ProcessorBuilder;

	while(m_FormatProtocol_InformationBase.getFormatAndProtocolVector(
		loc_nFormat, loc_ProtocolVector)){

// 4.  calls the function createDataFormatProcessor()
//     of CProcessorBuilder.  CMonitorManager passes an
//     int for the data format into this function.  This
//     function returns the data format processor,
//     CAbsDataFormatter, to CMonitorManager.

		CAbsDataFormatter * loc_pAbsDataFormatter =
			loc_ProcessorBuilder.createDataFormatProcessor(loc_nFormat);

// 5.  calls the function formatEventData() of
//     CAbsDataFormatter.  CMonitorManager passes the
//     usage information, CAbsEventData, into this
//     function.  This function returns the formatted
//     usage information, CAbsFormattedEventData, to
//     CMonitorManager.

		CAbsFormattedEventData * loc_pAbsFormattedEventData =
			loc_pAbsDataFormatter->formatEventData(loc_pAbsEventData);

// 6.  calls the function createProtocolProcessor() of
//     CProcessorBuilder.  CMonitorManager passes an int
//     for the communication protocol into this function.
//     The int is the first int from the protocol vector,
//     list<int>.  This function returns the protocol
//     processor, CAbsProtocolProcessor, to CMonitorManager.

		for(list<int>::iterator loc_ProtocolVectorIterator =
		   loc_ProtocolVector.begin();  loc_ProtocolVectorIterator NE
		    loc_ProtocolVector.end(); loc_ProtocolVectorIterator ++){
```

Figure16A

```
                        CAbsProtocolProcessor * loc_pAbsProtocolProcessor =
                              loc_ProcessorBuilder.createProtocolProcessor(
                                * loc_ProtocolVectorIterator);

//  7.  calls the function processFormattedData() of
//      CAbsProtocolProcessor.  CMonitorManager passes the
//      formatted usage information, CAbsFormattedEventData,
//      into this function.  This function returns a bool to
//      CMonitorManager to indicate if the usage information
//      was communicated using the protocol.

                                loc_pAbsProtocolProcessor->processFormattedData(
                                      loc_pAbsFormattedEventData);

                           }
//    8.  steps 6 and 7 are repeated for each protocol,
//        int, in the protocol vector, list<int>.
        }
//    9.  steps 3 through 8 are repeated for each format
//        until the function getFormatAndProtocolVector()
//        returns NO to CMonitorManager.
}
```

Figure 16B

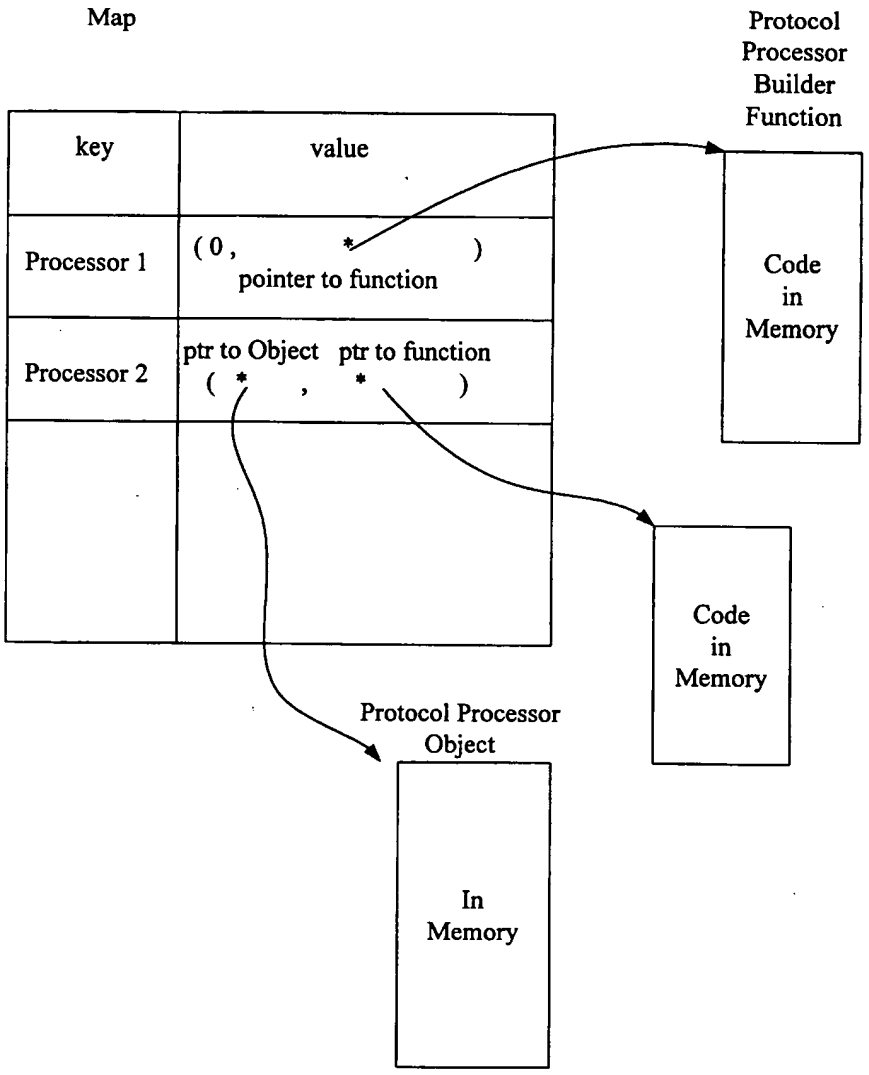

Map
Protocol Processor Builder Function
key
value
Processor 1
(0 , *)
pointer to function
Processor 2
ptr to Object ptr to function
(* , *)
Code in Memory
Code in Memory
Protocol Processor Object
In Memory

Figure 17

Author: Avery Fong
3.3 CProcessorBuilder Class Specification

3.3.1 Function List

```
public:
  CProcessorBuilder();
  ~CProcessorBuilder();
  CAbsDataFormatter* createDataFormatProcessor(int in_nFormat);
  CAbsProtocolProcessor* createProtocolProcessor(int in_nProtocol);

private:
  void initDataFormatProcessorMap();
  void initProtocolProcessorMap();
```

Include the following functions to create the different data format processors and protocol processors.

```
  CAbsDataFormatter* createCommaDataFormatter();
  CAbsDataFormatter* createXMLDataFormatter();
  CAbsProtocolProcessor* createSmtpProtocolProcessor();
  CAbsProtocolProcessor* createFtpProtocolProcessor();
```

If new data formats or new protocols are added, then new functions to create them must be added.

Include the following typedef declarations for the functions that create the data format processors and protocol processors.

```
  typedef CAbsDataFormatter* (*DataFormatProcessorBuilder) ();
  typedef CAbsProtocolProcessor* (*ProtocolProcessorBuilder) ();
```

3.3.2 Class Attributes

Type	Attribute Name	Description
CAbsDataFormatter*	m_pDataFormatter	This attribute member points to the data format processor object. It is initialize to 0 in the constructor and the data format processor object is created by the function createDataFormatProcessor(). This function may be called multiple times so that it must delete the previous data format processor object pointed to by this attribute member before creating a new one. The destructor will delete the last data format processor object pointed to by this attribute member.
map<int, DataFormatProcessorBuilder>	m_ProtocolProcessorMap	This attribute member is a map of pointers to functions that create the data format processor. The key to this map is an int for the data format type. The value is a pointer to a function that creates the data format processor corresponding to the key. The pointers to the functions in the map are initialized in the function initDataFormatProcessorMap().
map<int, pair<CAbsProtocolProcessor*, ProtocolProcessorBuilder> >	m_ProtocolProcessorMap	This attribute member is a map of pointers to protocol processor objects and pointers to functions that create them. The key to this map is an int for the protocol processor type. The value is a pair consisting of a pointer to the protocol processor object and a pointer to a function that creates the protocol processor object. All the pointers to the protocol processor object are initialized to 0 and its corresponding functions are intialized by the function initProtocolProcessorMap(). The protocol processor objects are created by the function createProtocolProcessor(). The destructor will delete all the protocol processor objects pointed to by the map.

Figure 18A

```
3.3.3    Function Definitions
////////////////////////////////////////////////////////////////////////////
// Function:          CProcessorBuilder
// Description:       Constructor
// Preconditions:     None.
// Postconditions:    None.
// Algorithm:         1.  calls the private function
//                    initDataFormatProcessorMap().
//                    2.  calls the private function
//                    initProtocolProcessorMap().
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
// Function:          ~CProcessorBuilder
// Description:       Destructor
// Preconditions:     None.
// Postconditions:    None.
// Algorithm:         1.  delete the object pointed to by m_pDataFormatter.
//                    2.  iterate through the map, m_ProtocolProcessorMap.
//                    For each entry in the map, get the protocol
//                    processor object pointed to by the pair and delete
//                    the object.
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
// Function:          createDataFormatProcessor
// Description:       This function creates a data format processor
//                    object.  The data format processor object created
//                    corresponds to the data format type in_nFormat.
// Preconditions:     The data format type must be valid.
// Postconditions:    The pointer to the data format processor object,
//                    m_pDataFormatter, cannot be 0.
// Algorithm:         1.  if m_pDataFormatter currently points to a data
//                    format processor object, then delete the object.
//                    2.  creates a new data format processor object by
//                    calling the function in the map,
//                    m_DataFormatProcessorMap, that corresponds to the
//                    data format type, in_nFormat, and assign it to
//                    m_pDataFormatter.
//                    3.  returns m_pDataFormatter.
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Function:          createProtocolProcessor
// Description:       This function creates a protocol processor object.
//                    The protocol processor object created corresponds
//                    to the protocol type in_nProtocol.
// Preconditions:     The protocol type must be valid.
// Postconditions:    The pointer to the created protocol processor object
//                    cannot be 0.
// Algorithm:         1.  for the protocol type, in_nProtocol, get the
//                    pair from the map that contains the pointer to
//                    protocol processor object and its corresponding
//                    pointer to the function that creates it.
//                    2.  if the pointer to the protocol processor object
//                    is 0, then use its corresponding function to create
//                    it and assign it to the pointer in the map.  Return
//                    the pointer to the protocol processor object.
//                    3.  if the pointer points to a protocol processor
//                    object, then return this pointer.
////////////////////////////////////////////////////////////////////////////
```

Figure18B

```
////////////////////////////////////////////////////////////////////////
// Private
// Function:          initDataFormatProcessorMap
// Description:       This function initializes all the function pointers
//                    in the map m_DataFormatProcessorMap.  If new data
//                    formats are added, then this function must be
//                    modified.
// Preconditions:     None.
// Postconditions:    None.
// Algorithm:         1.  add entries to the map, m_DataFormatProcessorMap,
//                    for each data format type.  The key will be the
//                    data format type and the value will be the pointer
//                    to the corresponding function that creates the
//                    data format processor.
//                    2.  for data format type 1, the function pointer
//                    points to createCommaDataFormatter().
//                    3.  for data format type 2, the function pointer
//                    points to createXMLDataFormatter().
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
// Private
// Function:          initProtocolProcessorMap
// Description:       This function initializes all the pairs of pointers
//                    in the map m_ProtocolProcessorMap.  If new protocols
//                    are added, then this function must be modified.
// Preconditions:     None.
// Postconditions:    None.
// Algorithm:         1.  add entries to the map, m_ProtocolProcessorMap,
//                    for each protocol type.  The key will be the
//                    protocol type and the value will be a pointer to
//                    the protocol processor object and a pointer
//                    to the corresponding function that creates the
//                    protocol processor.  All pointers to the protocol
//                    processor objects will be set to 0.
//                    2.  for protocol type 1, the function pointer
//                    points to createSmtpProtocolProcessor().
//                    3.  for protocol type 2, the function pointer
//                    points to createFtpProtocolProcessor().
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
// Function:          createCommaDataFormatter
// Description:       This function creates and returns a comma data
//                    formatter object.
// Preconditions:     None.
// Postconditions:    The pointer to the created comma data formatter
//                    object cannot be 0.
// Algorithm:         1.  creates and returns an object of the class
//                    CCommaDataFormatter.
////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////
// Function:          createXMLDataFormatter
// Description:       This function creates and returns an XML data
//                    formatter object.
// Preconditions:     None.
// Postconditions:    The pointer to the created XML data formatter
//                    object cannot be 0.
// Algorithm:         1.  creates and returns an object of the class
//                    CXMLDataFormatter.
////////////////////////////////////////////////////////////////////////
```

Figure 18C

```
////////////////////////////////////////////////////////////////////////
// Function:          createSmtpProtocolProcessor
// Description:       This function creates and returns an SMTP protocol
//                    processor object.
// Preconditions:     None.
// Postconditions:    The pointer to the created smtp protocol processor
//                    object cannot be 0.
// Algorithm:         1.  creates and returns an object of the class
//                    CSmtpProtocolProcessor.
////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////
// Function:          createFtpProtocolProcessor
// Description:       This function creates and returns an FTP protocol
//                    processor object.
// Preconditions:     None.
// Postconditions:    The pointer to the created ftp protocol processor
//                    object cannot be 0.
// Algorithm:         1.  creates and returns an object of the class
//                    CFtpProtocolProcessor.
////////////////////////////////////////////////////////////////////////
```

Figure 18D

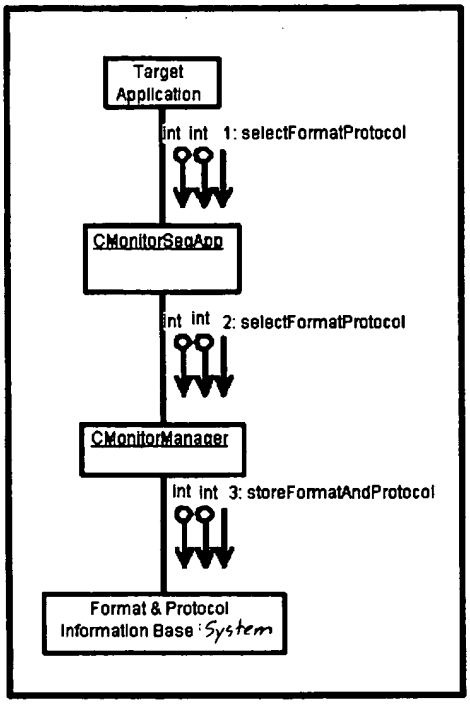
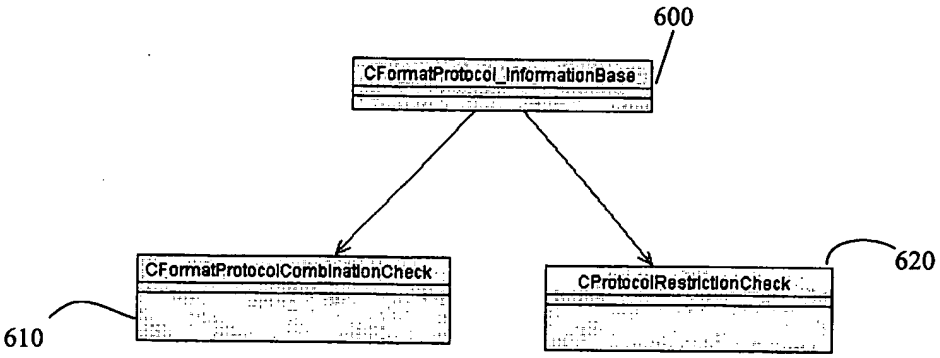

Target Application
int int 1: selectFormatProtocol
CMonitorSapApp
int int 2: selectFormatProtocol
CMonitorManager
int int 3: storeFormatAndProtocol
Format & Protocol Information Base : System

Figure 19



Format And Protocol Information Base Package Class Structure

Figure 20

DOCKET NO. 50277-0005

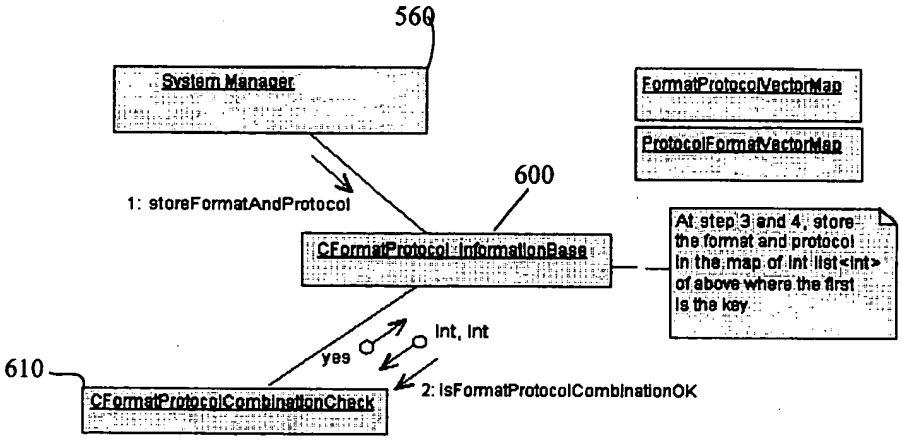


Figure 21

0704.0536.0US.WO

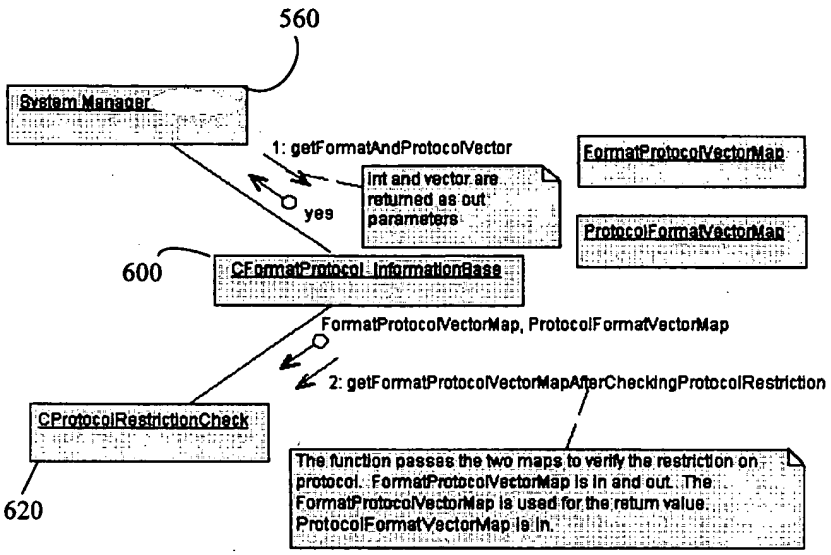

560
System Manager
1: getFormatAndProtocolVector
yes
int and vector are returned as out parameters
FormatProtocolVectorMap
ProtocolFormatVectorMap
600
CFormatProtocol_InformationBase
FormatProtocolVectorMap, ProtocolFormatVectorMap
2: getFormatProtocolVectorMapAfterCheckingProtocolRestriction
CProtocolRestrictionCheck
620
The function passes the two maps to verify the restriction on protocol. FormatProtocolVectorMap is in and out. The FormatProtocolVectorMap is used for the return value. ProtocolFormatVectorMap is in.

Figure 22

Author: Tetsuro Motoyama

5.2 CFormatProtocol_InformationBase Class Specification

5.2.1 Function List

```
public:
  CFormatProtocol_InformationBase();
  ~CFormatProtocol_InformationBase();
  void storeFormatAndProtocol(int in_nFormat, int in_nProtocol);
  bool getFormatAndProtocolVector(int & out_nFormat, list<int> & out_ProtocolVector);

private:
  void setDefaultFormatAndProtocol();
```

5.2.2 Class Attributes

Type	Attribute Name	Description
map<int, list<int> >	m_FormatProtocolVectorMap	The key is a format value, and the list is the list of protocol values associated to the key. Because subscripting [] is not needed in this implementation, list is used for the vector implementation. This map is used to return the necessary information for getFormatAndProtocolVector function Note: > > is > space > to distinguish from ">>" that is used by iostream.
map<int, list<int> >	m_ProtocolFormatVectorMap	The key is a protocol value, and the list is the list of format values associated to the key. Because subscripting [] is not needed in this implementation, list is used for the vector implementation. This map is used to modify the map above if the protocol can take only one format.
bool	m_bFirstGetCall	This flag is used to call the function in CProtocolRestrictionCheck. The constructor set this to be true. The function, getFormatAndProtocolVector, sets it to be false
map<int, list<int> >::iterator	m_FormatProtocolVectorMapIterator	Iterator used to iterate the map.
CFormatProtocolCombinationCheck	m_FormatProtocolCombinationCheck	This object is to check the combination of format and protocol
CProtocolRestrictionCheck	m_ProtocolRestrictionCheck	This object is to check the protocol restriction. Currently, the only restriction is if protocol can have only one format support.

5.2.3 Function Definitions

```
////////////////////////////////////////////////////////////////////
//Function:          CFormatProtocol_InformationBase
//Description:       Constructor
//Preconditions:     None
//Postconditions:    None
//Algorithm:         Set m_bFirstGetCall to true
////////////////////////////////////////////////////////////////////
```

Figure 17A
23A

```
////////////////////////////////////////////////////////////////////
//Function:          ~CFormatProtocol_InformationBase
//Description:       Destructor
//Preconditions:     None
//Postconditions:    None
//Algorithm:         Default
////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////
//Function:           storeFormatAndProtocol
//Description:        Check the passed format and protocol values
//                    to be valid or not.  If valid, store the
//                    values into the two maps
//Preconditions:      None
//Postconditions:     None
//Algorithm:          1. Send two values to check the combination
//                       through isFormatProtocolCombinationOK
//                       function.
//                    2. Check the return bool value.
//                    3. If yes, save format and protocol values
//                         into two maps  (Figure 5.4 of the
//                         Specification, Q6-DJ04-08)
//                       Else, do nothing.
////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////
//Function:           getFormatAndProtocolVector
//Description:        The function returns a format and a list
//                    of protocol values associated with the
//                    format through two parameters.  The function
//                    returns true if a format and list are
//                    returned, false otherwise.
//Preconditions:      None
//Postconditions:     The format value is within the range.
//                    The list is not empty and contains the values
//                    within the range.
//Algorithm:          1. If m_bFirstGetCall  (Figure 5.5 of the
//                          Specification Q6-DJ04-08)
//                       1.1 call the function to check the protocol
//                           restriction.
//                       1.2 check if m_FormatProtocolVectorMap is
//                           empty.  If empty, set it to default
//                           values of format and protocol by calling
//                           setDefaultFormatAndProtocol function.
//                       1.3 set the iterator to begin().
//                       1.4 set m_bFirstGetCall to be false
//                    2. If iterator is end, return false.
//                       else (Figure 5.6 of the Specification
//                            Q6-DJ04-08)
//                         get format and list to return and set
//                         return parameters.
//                         increment iterator.
//                         Return true.
////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////
//Private Function:   setDefaultFormatAndProtocol
//Description:        The function sets the default values for
//                    format and protocol in the map.
//Preconditions:      The m_FormatProtocolVectorMap is empty.
//Postconditions:     The map contains one default format and a
//                    protocol list with one default protocol.
//Algorithm:          Set the map with the default values.
////////////////////////////////////////////////////////////////
```

Figure 19R 23B

Author: Tetsuro Motoyama

5.3 CFormatProtocolCombinationCheck Class Specification

5.3.1 Function List

```
public:
  CFormatProtocolCombinationCheck();
  ~CFormatProtocolCombinationCheck()
  bool isFormatProtocolCombinationOK(const int in_nFormat, const int in_nProtocol);

private:
  void initMatrix();
```

5.3.2 Class Attributes

Type	Attribute Name	Description
map<int, set<int> >	m_CombinationMatrix	Key is the format. The set contains the protocols that are valid for the particular format

5.3.3 Function Definitions

```
////////////////////////////////////////////////////////////////////
//Function:          CFormatProtocolCombinationCheck
//Description:       Constructor
//Preconditions:     None
//Postconditions:    None
//Algorithm:         call initMatrix
////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////
//Function:          ~CFormatProtocolCombinationCheck
//Description:       Destructor
//Preconditions:     None
//Postconditions:    None
//Algorithm:         Default
////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////
//Function:          isFormatProtocolCombinationOK
//Description:       Check the passed format and protocol values
//                   to be valid or not.  If valid, return yes,
//                   no otherwise
//Preconditions:     None
//Postconditions:    None
//Algorithm:         1. Use find function of the Matrix for
//                      in_nFormat
//                   2. If returned iterator is end, return No
//                   3. get the set value for the key format
//                   4. Use the find function of the set for
//                      in_nProtocol
//                   5. if returned iterator is end, return no
//                   6. return yes
////////////////////////////////////////////////////////////////////
```

Figure 19A
24A

```
////////////////////////////////////////////////////////////////////
//Private Function:  initMatrix
//Description:       This function initializes m_CombinationMatrix.
//                   If new formats or protocols are added, this
//                   function must be modified.
//Precondition:      None
//Postconidition:    None
//Algorithm:         1. Create the local set<int>
//                   2  for each format
//                        2.1 fill in the local set
//                            with the protocol numbers
//                            that are valid for the format,
//                            using insert function
//                        2.2 m_CombinationMatrix[format]
//                              = local set
//                        2.3 clear local set
////////////////////////////////////////////////////////////////////
```

Figure 15B

Author: Tetsuro Motoyama

5.4 CProtocolRestrictionCheck Class Specification

5.4.1 Function List

```
public:
  CProtocolRestrictionCheck();
  ~CProtocolRestrictionCheck()
  void getFormatProtocolVectorMapAfterCheckingProtocolRestriction
      (map<int, list<int> > & inOut_Map, const map<int, list<int> > & in_Map);

private:
  void initOneFormatRestriction();
  void oneFormatRestriction
      (map<int, list<int> > & inOut_Map, const map<int, list<int > > & in_Map);
```

5.4.2 Class Attributes

Type	Attribute Name	Description
vector<bool>	m_bOneFormatRestriction	Array size should be protocol size+1. The position corresponds to the protocol.

5.4.3 Function Definitions

```
////////////////////////////////////////////////////////////////////
//Function:          CProtocolRestrictionCheck
//Description:       Constructor
//Preconditions:     None
//Postconditions:    None
//Algorithm:         call initOneFormatRestriction
////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////
//Function:          ~CProtocolRestrictionCheck
//Description:       Destructor
//Preconditions:     None
//Postconditions:    None
//Algorithm:         Default
////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////
//Function:          getFormatProtocolVectorMapAfterCheckingProtocolRestriction
//Description:       Check the restriction on the protocol.
//                   Currently, there is only one possible restriction
//                   defined in the requirements.  If there are more
//                   restrictions, more private functions should be
//                   added and called.
//Preconditions:     None
//Postconditions:    None
//Algorithm:         1. Call oneFormatRestriction function
////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////
```

Figure 19A
25A

```
//Private Function:  initOneFormatRestriction
//Description:       This function initialize the attribute
//                   m_bOneFormatRestriction.  If more protocols are
//                   added, this initialization must be modified.
//Preconditions:     None
//Postconditions:    None
//Algorithm:         1. use assign(size+1,false) to initialize the
//                      vector to false.
//                   2. set the entries of true.
//                   Note: for class debug version, use
//                         ifdef and
//                         bool & pos1 = m_bOneFormatRestriction[1];
//                         bool & pos2 = m_bOneFormatRestriction[2];
//                         and so on to be able to see and to
//                         change the value.
////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////
//Private Function:  oneFormatRestriction
//Description:       This function receives two maps and if the one
//                   restriction is true for given protocol, the
//                   content of inOut_Map (m_FormatProtocolVectorMap)
//                   is adjusted accordingly.
//Preconditions:     None
//Postconditions:    None
//Algorithm:         Iterate over the in_Map (m_ProtocolFormatVectorMap)
//                   1. get the key (pkey)
//                   2. If m_bOneFormatRestriction[pkey]
//                      2.1 get the value list of in_Map for the key
//                      2.2 local int lastFormat = back(),
//                      2.3 iterate over the list
//                           if *iterator NE lastFormat
//                             iterate over inOut_Map[*iterator] list
//                                 if the value EQ pkey
//                                     erase the entry from the list
//                   3. Iterate over inOut_Map
//                        if value list is empty,
//                          erase the entry from inOut_Map
//--------------------------------------------------------------------
//Example:                          0 1 2 3 4
//   m_bOneFormatRestriction = [0,0,1,0,1]  (four protocols)
//                                0: false, 1: true
//   inOut_Map (m_FormatProtocolVectorMap)
//      = { 1, <1,2,3,4>                --> <1, 2, 3>
//          2, <2,1,3,4>                --> <1, 3>
//          3, <3,4,1,2>                --> <3, 4, 1>
//          4, <2,4>)                   --> <>
//   in_Map (m_ProtocolFormatVectorMap)
//      = (1, <1, 3, 2>
//         2, <4, 3, 2, 1>
//         3, <1, 3, 2>
//         4, <4, 2, 1, 3>)
//
//   pkey = 1  m_bOneFormatRestriction[1] = 0
//   pkey = 2  m_bOneFormatRestriction[2] = 1
//     value list = <4, 3, 2, 1>  (2.1)
//     lastFormat = 1             (2.2)
//     4 != 1
//         inOut_Map[4]  = <2,4>
//         erase value 2   <4>
//     3 != 1
//         inOut_Map[3]  = <3,4,1,2>
//         erase value 2   <3,4,1>
//     2 !=1
//         inOut_Map[2]  = <2,1,3,4>
//         erase value 2   <1,3,4>
//     1 == 1
//   pkey = 3  m_bOneFormatRestriction[3] = 0
```

Figure 19B
25B

```
//   pkey = 4  m_bOneFormatRestriction[4] = 1
//     value list = <4, 2, 1, 3>
//     lastFormat = 3
//     4 != 3
//         inOut_Map[4] = <4>
//         erase value 4  <>
//     2 != 3
//         inOut_Map[2]  = <1,3,4>
//         erase value 4   <1,3>
//     1 != 3
//         inOut_Map[1]  = <1,2,3,4>
//         erase value 4   <1,2,3>
//     3 == 3
//    Iterate over inOut_Map
//       if *inOut_Map_iterator.empty() then erase
//
//    inOut_Map
//        = ( 1, <1, 2, 3>
//            2, <1, 3>
//            3, <3, 4, 1>)
//
////////////////////////////////////////////////////////////////
```

Figure 19C
25C